dongguk UNIVERSITY

CSE 2017 Data Structures and Lab

Lecture #5: Queue

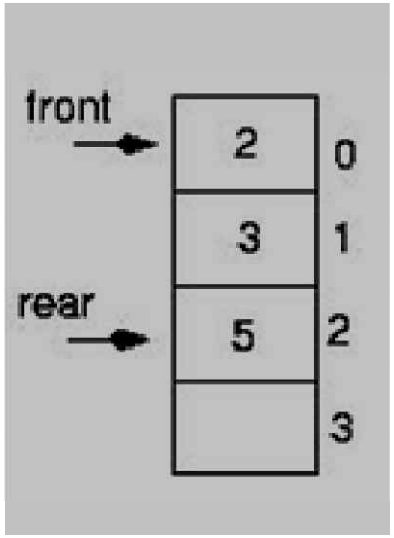
Eun Man Choi

What is a queue?

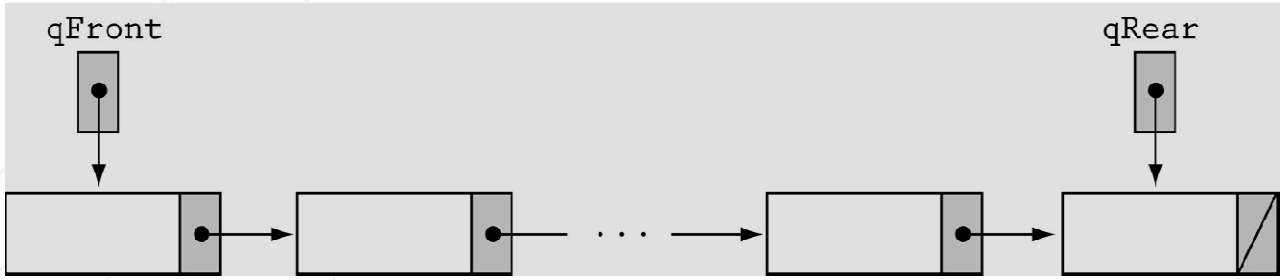
- It is an ordered group of homogeneous items.
- Queues have two ends:
  - Items are added at one end.
  - Items are removed from the other end.
- FIFO property: First In, First Out
  - The item added first is also removed first

Queue Implementations

Array-based

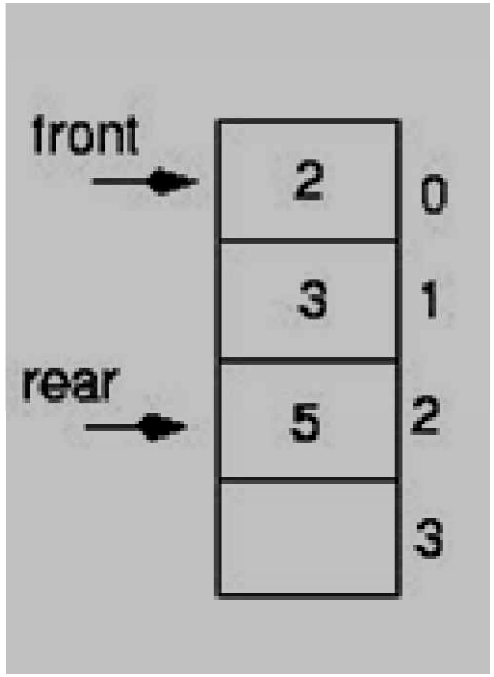


Linked-list-based



Array-based Implementation

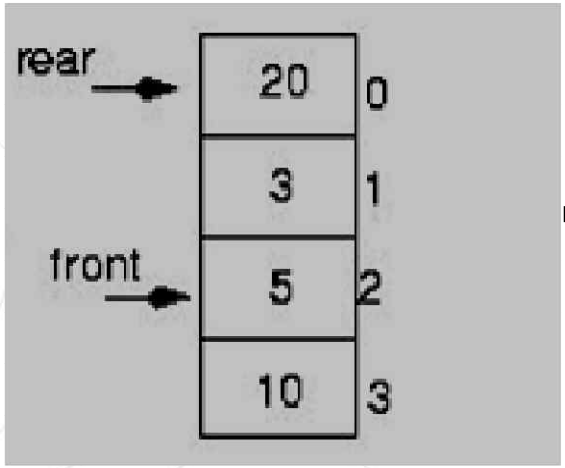
```
template<class ItemType>
class QueueType {
 public:
    QueueType(int);
     ~QueueType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Enqueue(ItemType);
    void Dequeue(ItemType&);
 private:
    int front, rear;
    ItemType* items;
    int maxQue;
};
```



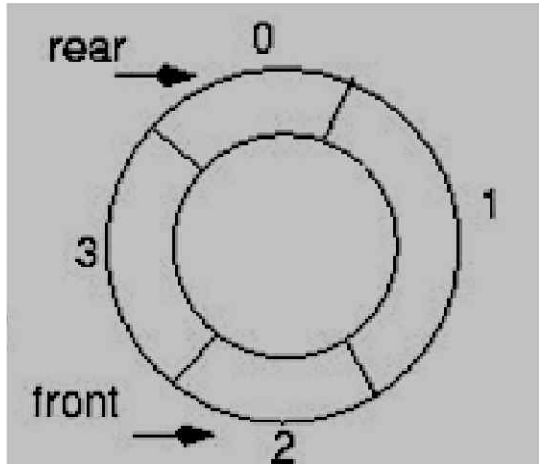
Implementation Issues

- *Optimize* memory usage.
- Conditions for a *full* or *empty* queue.
- Initialize *front* and *rear*.

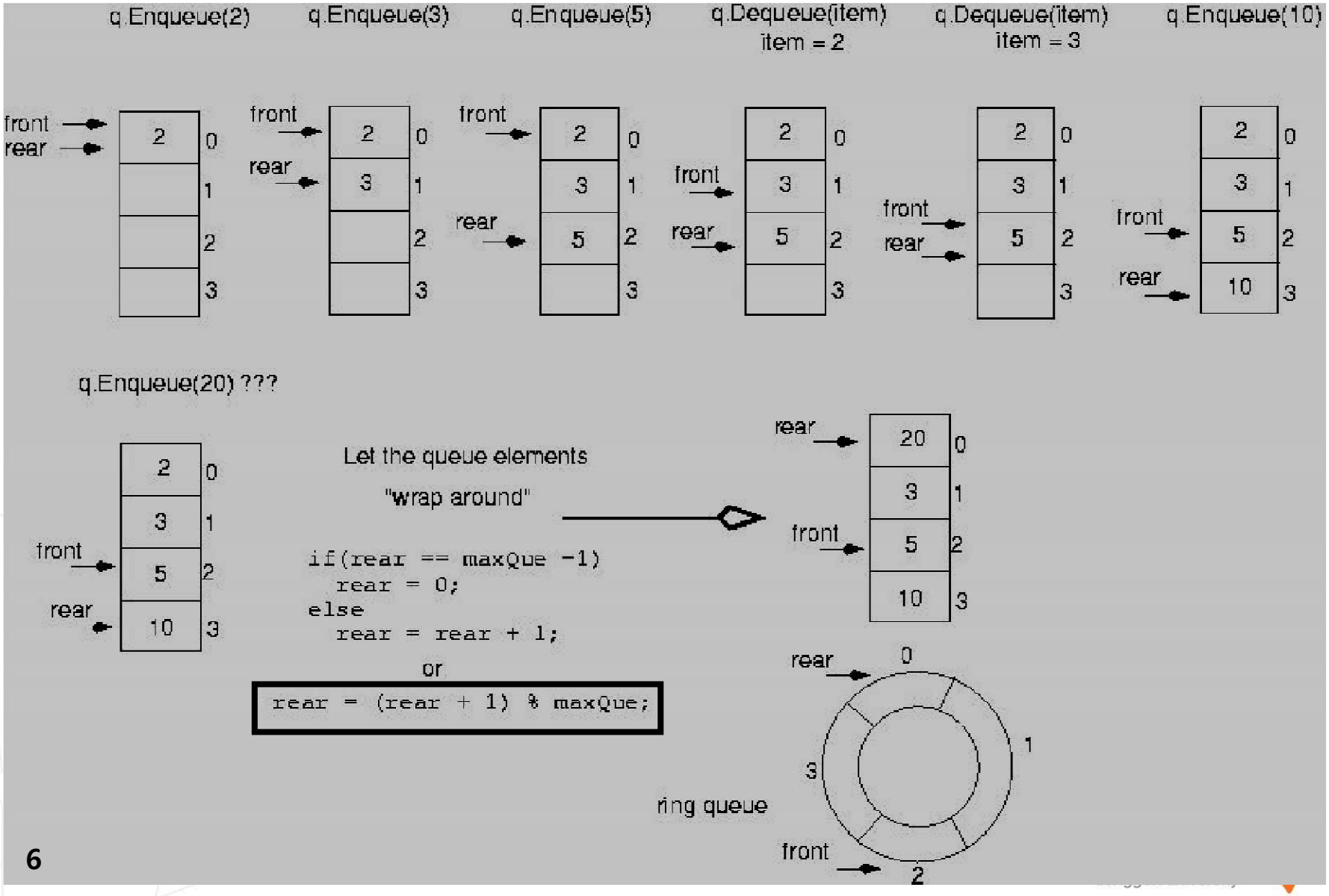
linear array



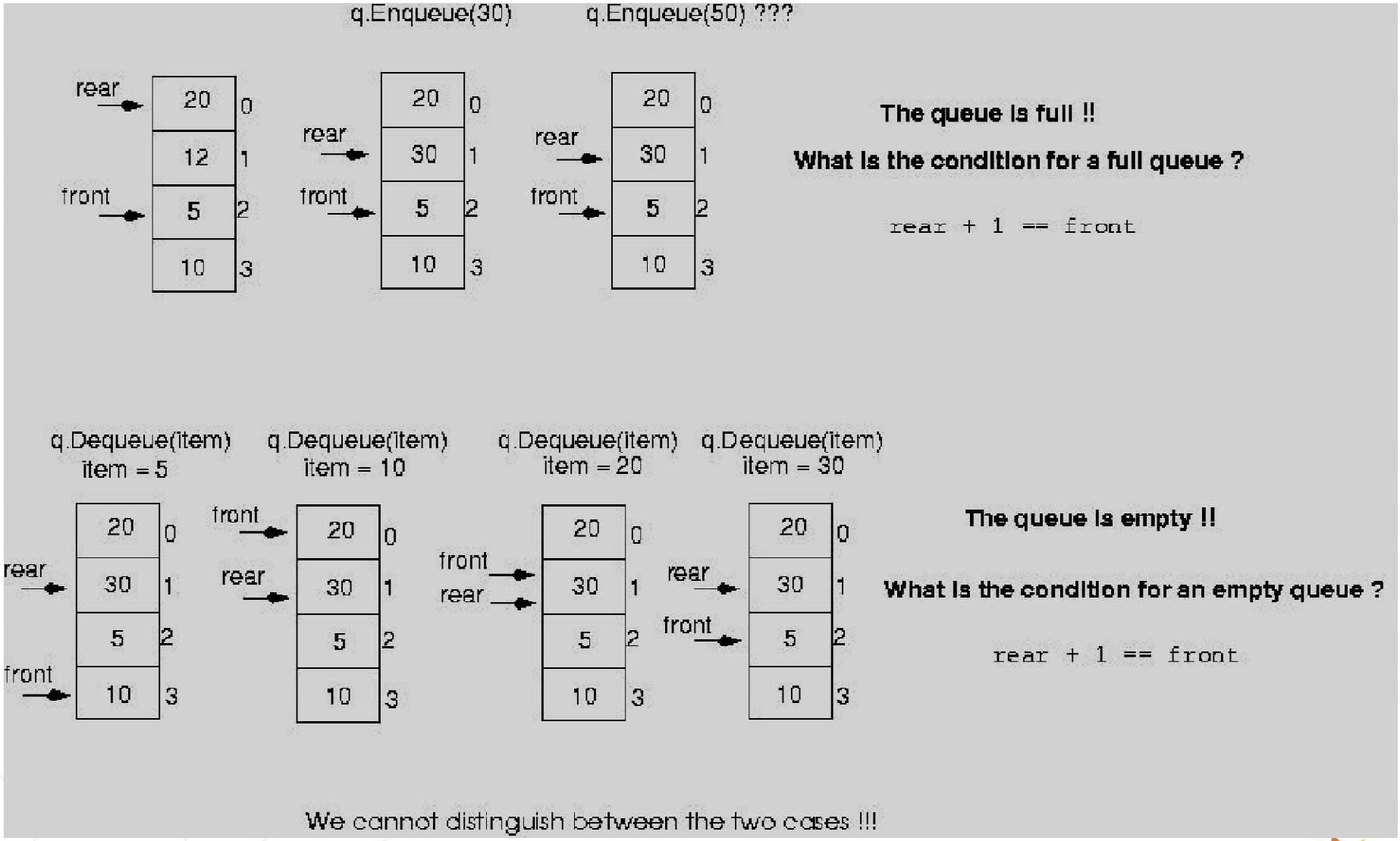
circular array



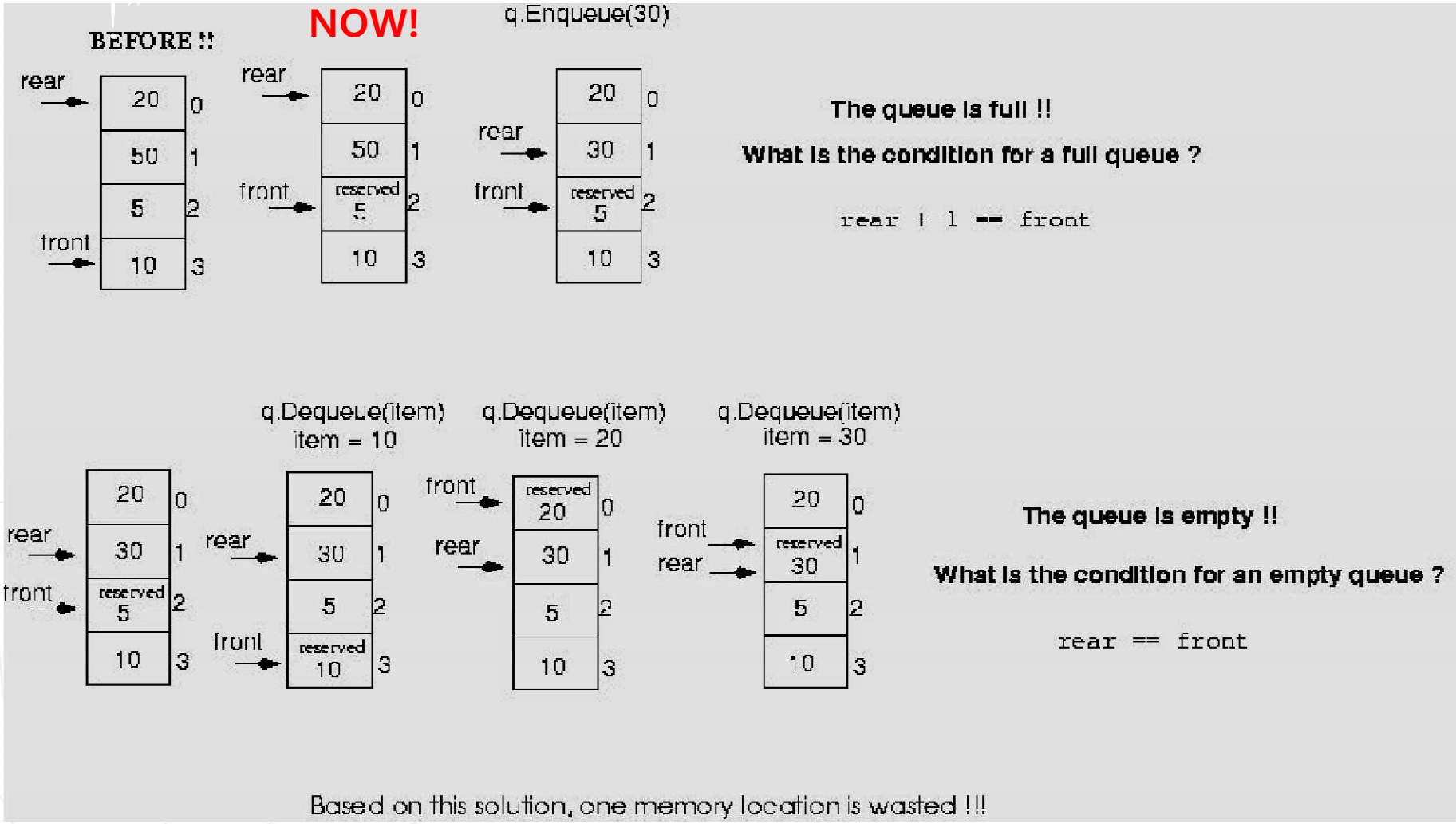
Optimize memory usage



Full/Empty queue conditions

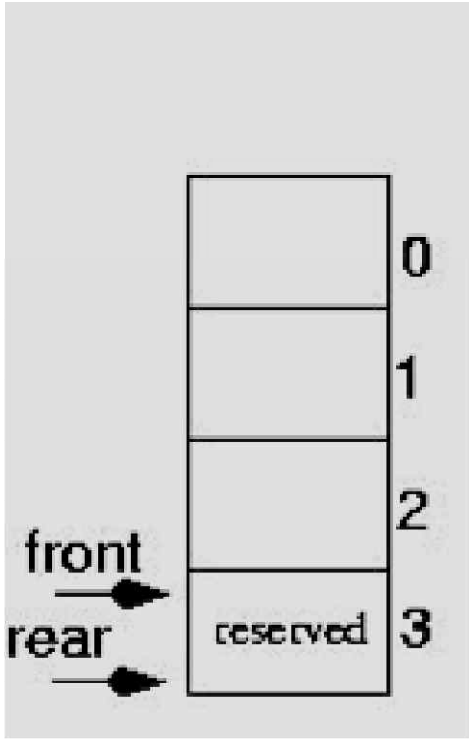


"Make *front* point to the element preceding the front element in the queue!"



Initialize *front* and *rear*

- Make *front* point to the element preceding the front element in the queue!



Array-based Implementation (cont.)

```
template<class ItemType>
QueueType<ItemType>::QueueType(int max)
{
 maxQue = max + 1;
 front = maxQue - 1;
 rear = maxQue - 1;
 items = new ItemType[maxQue];
}
```

O(1)

```
template<class ItemType>
QueueType<ItemType>::~QueueType()
{
 delete [] items;
}
```

O(1)

Array-based Implementation (cont.)

```
template<class ItemType>
void QueueType<ItemType>::MakeEmpty()
{
 front = maxQue - 1;
 rear = maxQue - 1;
}
```

O(1)

```
template<class ItemType>
bool QueueType<ItemType>::IsEmpty() const
{
 return (rear == front);
}
```

O(1)

Array-based Implementation (cont.)

```
template<class ItemType>
bool QueueType<ItemType>::IsFull() const
{
 return ( (rear + 1) % maxQue == front);
}
```

O(1)

Enqueue (ItemType newItem)

- *Function*: Adds newItem to the rear of the queue.
- *Preconditions*: Queue has been initialized and is not full.
- *Postconditions*: newItem is at rear of queue.

Queue overflow

- The condition resulting from trying to add an element onto a full queue.

```
if(!q.IsFull())
  q.Enqueue(item);
```

```
template<class ItemType>
void QueueType<ItemType>::Enqueue (ItemType newItem)
{
 rear = (rear + 1) % maxQue;
 items[rear] = newItem;
}
```

O(1)

Dequeue (ItemType& item)

- *Function*: Removes front item from queue and returns it in item.
- *Preconditions*: Queue has been initialized and is not empty.
- *Postconditions*: Front element has been removed from queue and item is a copy of removed element.

Queue underflow

- The condition resulting from trying to remove an element from an empty queue.

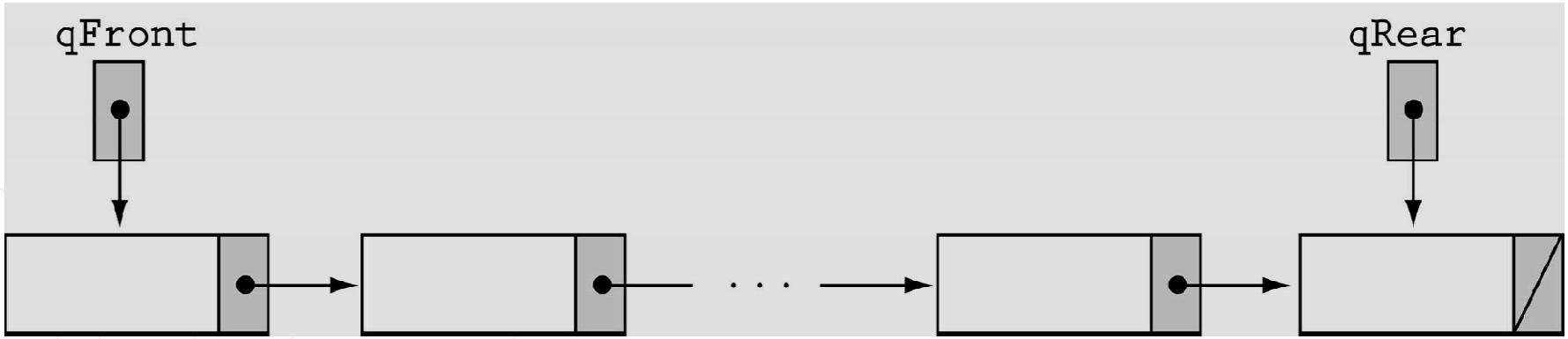
```
if(!q.IsEmpty())
  q.Dequeue(item);
```

```
template<class ItemType>
void QueueType<ItemType>::Dequeue (ItemType& item)
{
 front = (front + 1) % maxQue;
 item = items[front];
}
```

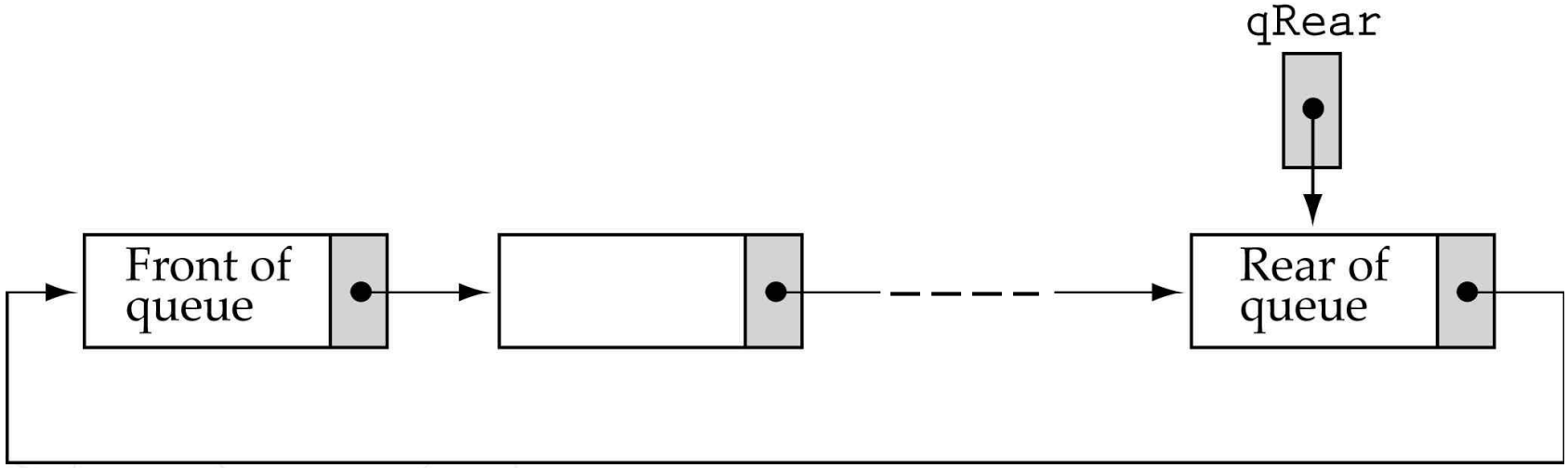
O(1)

Linked-list-based Implementation

- Allocate memory for each new element dynamically
- Link the queue elements together
- Use two pointers, *qFront* and *qRear*, to mark the front and rear of the queue

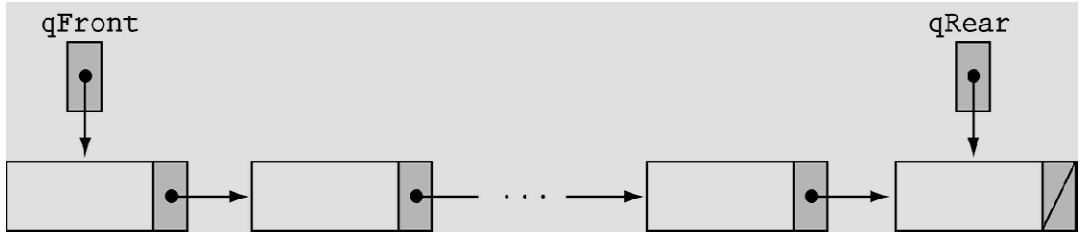


A “circular” linked queue design



Linked-list-based Implementation

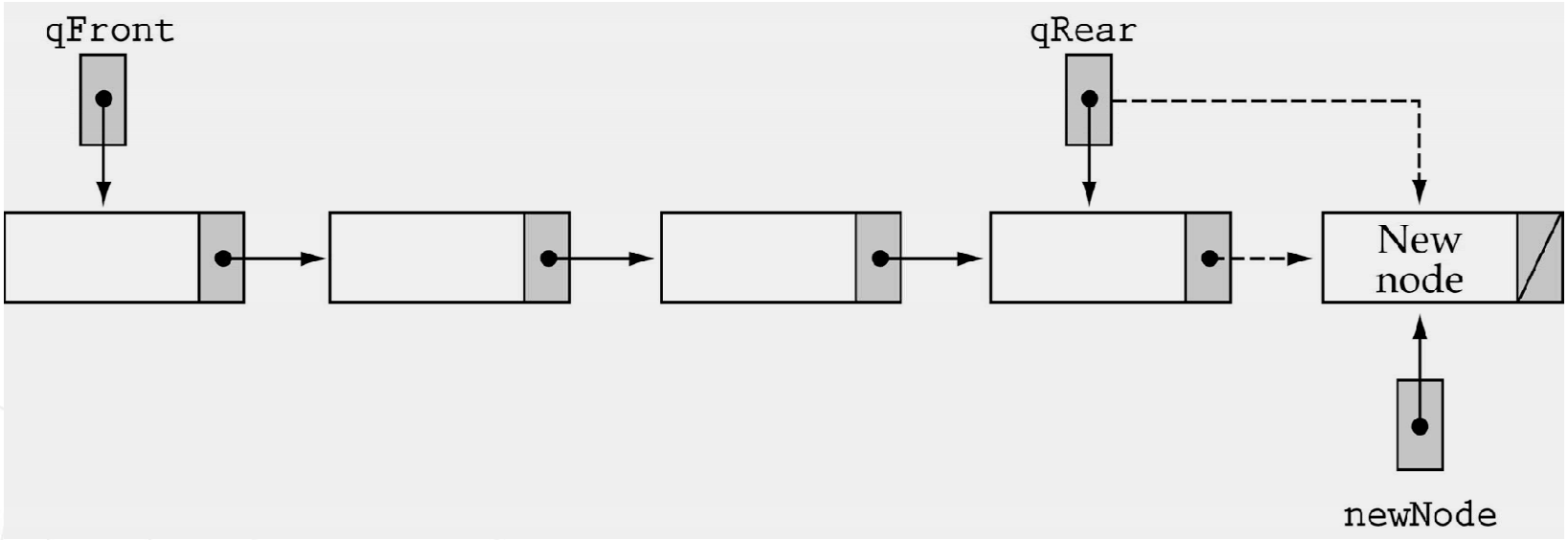
```
// forward declaration of NodeType (i.e., like function
  prototype)
template<class ItemType>
struct NodeType;

template<class ItemType>
class QueueType {
 public:
    QueueType();
    ~QueueType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Enqueue(ItemType);
    void Dequeue(ItemType&);
 private:
    NodeType<ItemType>* qFront;
    NodeType<ItemType>* qRear;
};
```



Enqueue (ItemType newItem)

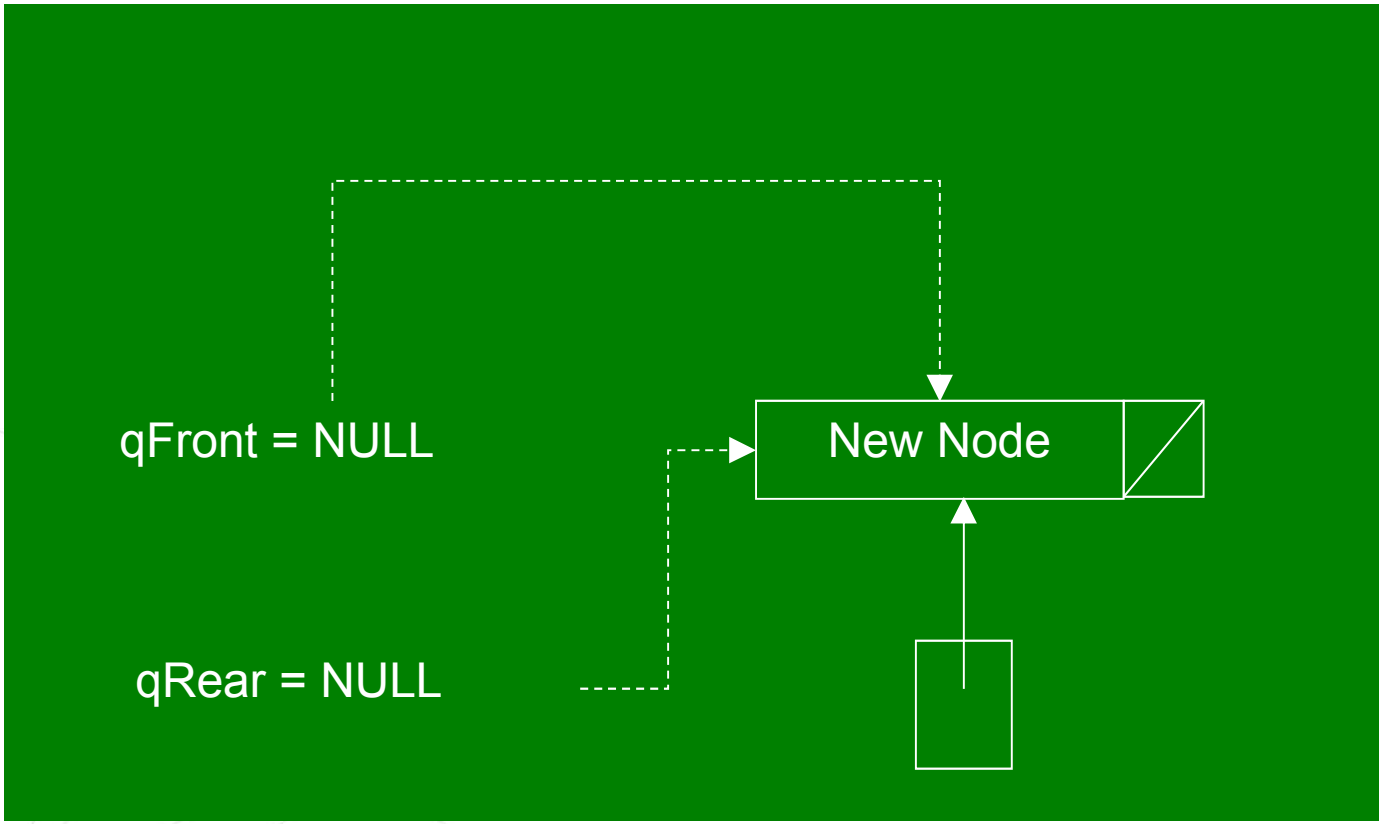
- *Function*: Adds newItem to the rear of the queue.
- *Preconditions*: Queue has been initialized and is not full.
- *Postconditions*: newItem is at rear of queue.

Enqueue (non-empty queue)



Special Case: empty queue

- **Need to make *qFront* point to the new node**



Enqueue

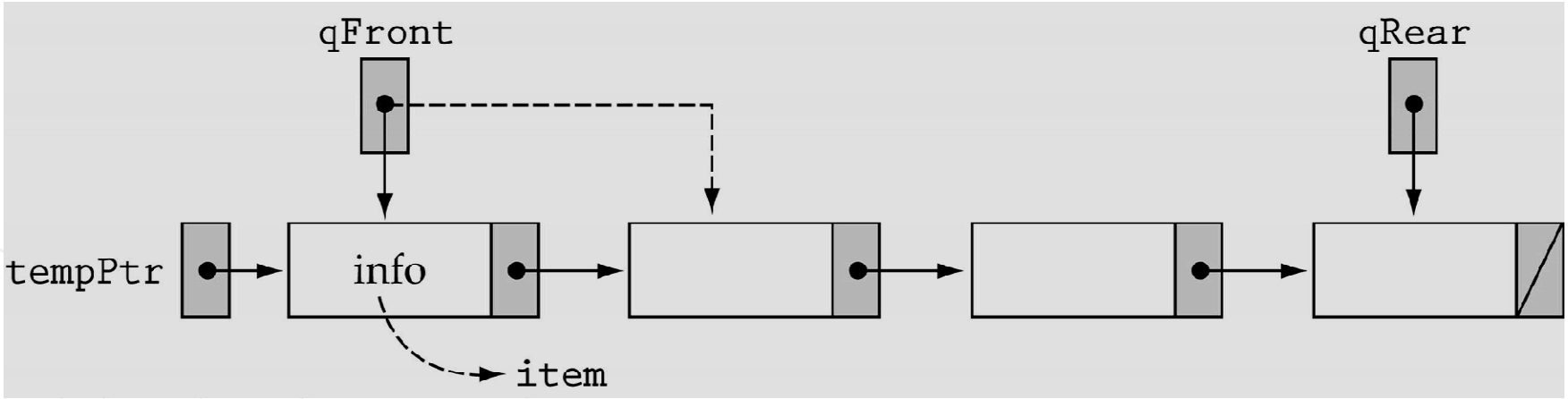
```
template <class ItemType>
void QueueType<ItemType>::Enqueue(ItemType newItem)
{
 NodeType<ItemType>* newNode;

 newNode = new NodeType<ItemType>;
 newNode->info = newItem;
 newNode->next = NULL;
 if(qRear == NULL) // special case
   qFront = newNode;
 else
   qRear->next = newNode;
 qRear = newNode;
}
```

O(1)

Dequeue (ItemType& item)

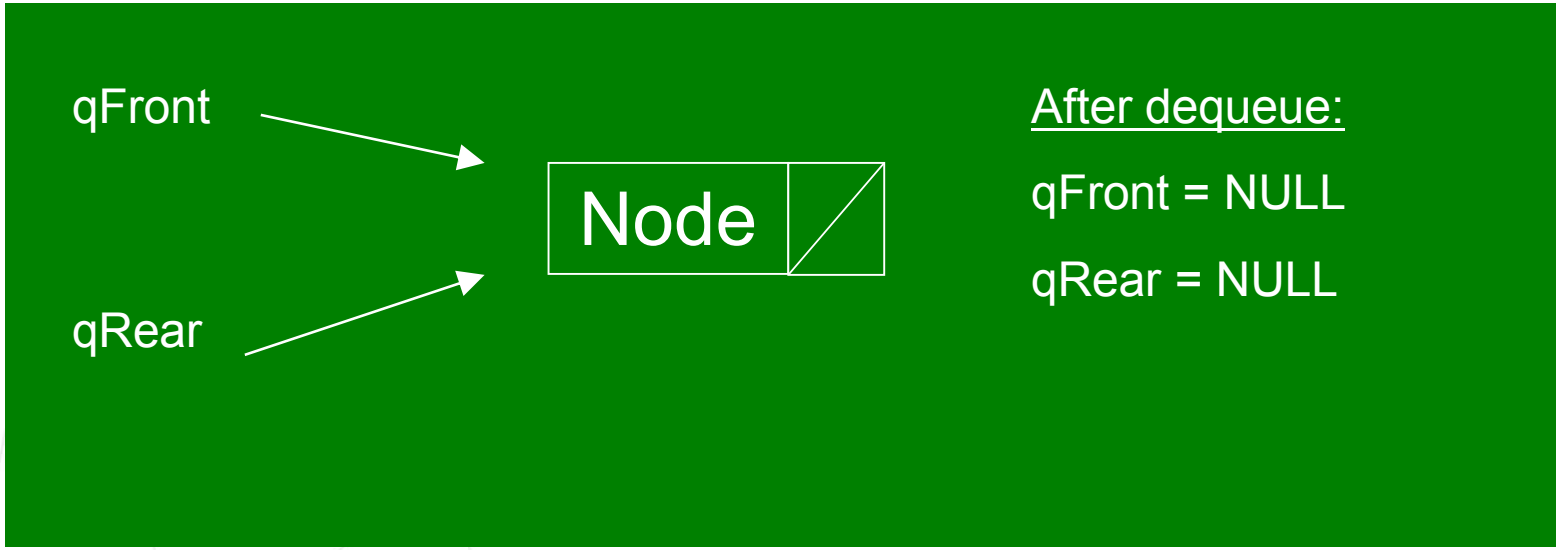
- *Function*: Removes front item from queue and returns it in item.
- *Preconditions*: Queue has been initialized and is not empty.
- *Postconditions*: Front element has been removed from queue and item is a copy of removed element.

Dequeue (queue contains more than one elements)



Special Case: queue contains a single element

- **We need to reset *qRear* to NULL also**



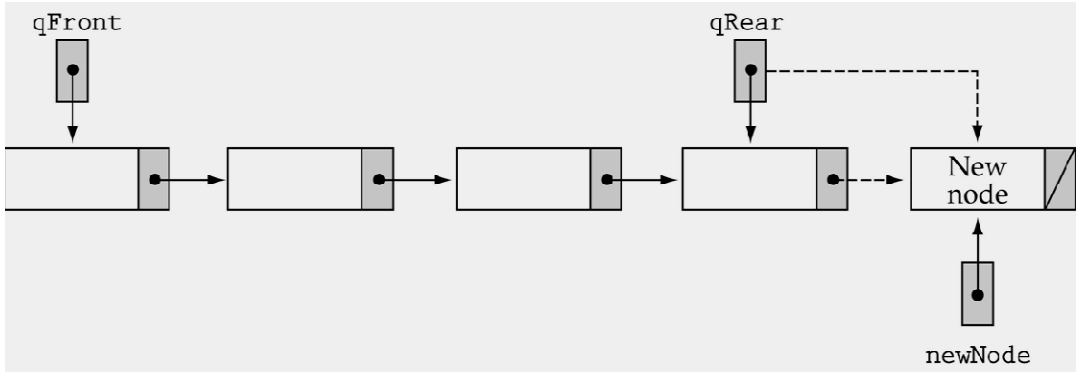
Dequeue

```
template <class ItemType>
void QueueType<ItemType>::Dequeue(ItemType& item)
{
 NodeType<ItemType>* tempPtr;

 tempPtr = qFront;
 item = qFront->info;
 qFront = qFront->next;
 if(qFront == NULL) // special case
   qRear = NULL;
 delete tempPtr;
}
```

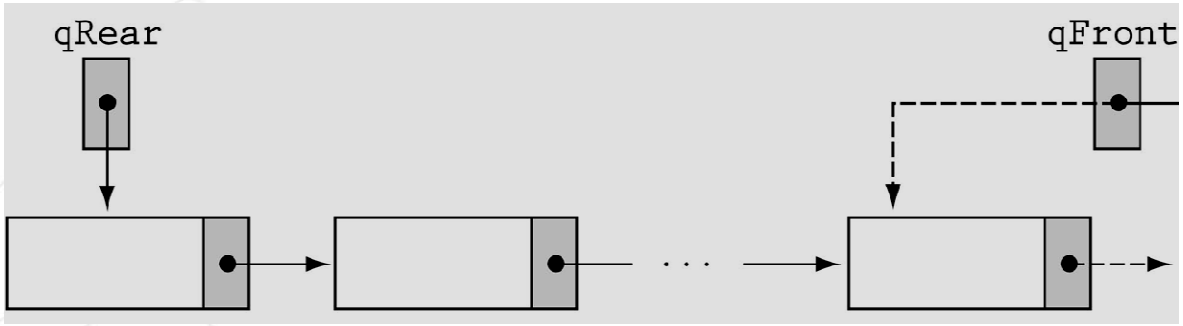
O(1)

qRear, qFront - revisited

- **Are the relative positions of *qFront* and *qRear* important?**

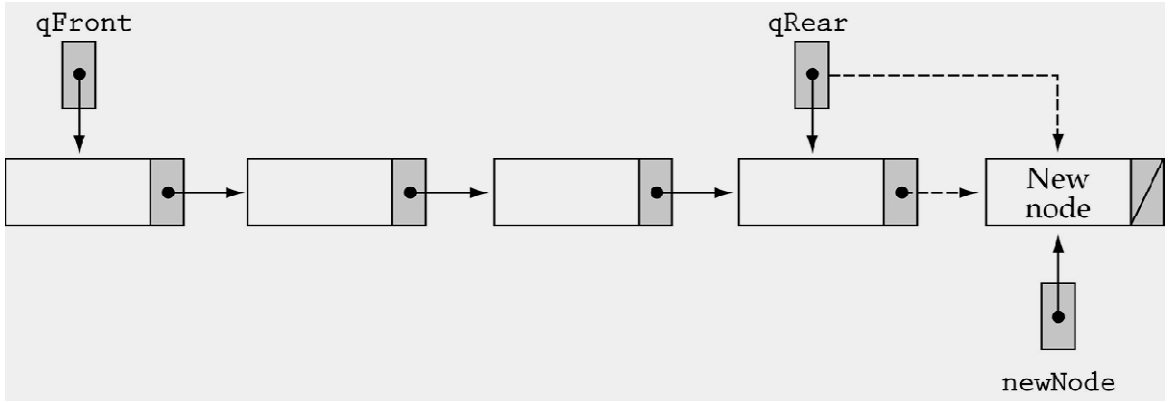


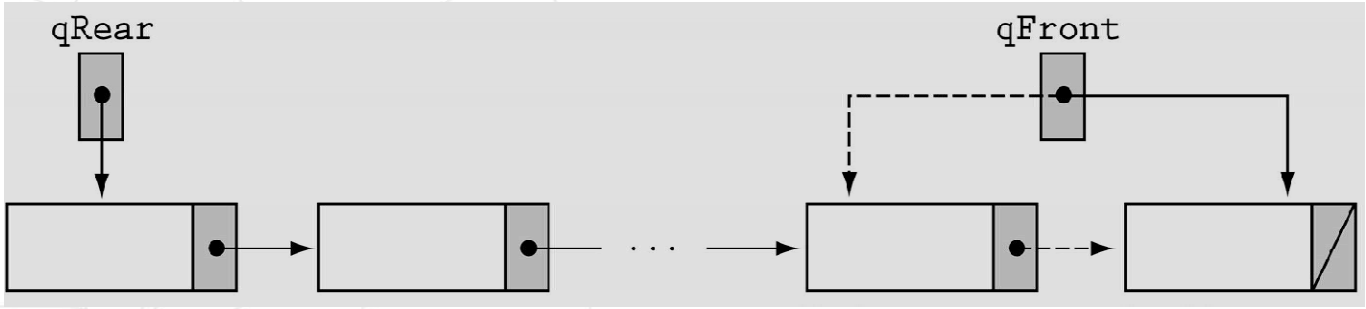
What about this?



qRear, qFront - revisited

- **Are the relative positions of *qFront* and *qRear* important?**





Hard to dequeue!

Other Queue functions

```
template<class ItemType>
QueueType<ItemType>::QueueType()
{
 qFront = NULL;
 qRear = NULL;
}
```

O(1)

```
template<class ItemType>
void QueueType<ItemType>::MakeEmpty()
{
 NodeType<ItemType>* tempPtr;

 while(qFront != NULL) {
   tempPtr = qFront;
   qFront = qFront->next;
   delete tempPtr;
 }
 qRear=NULL;
}
```

O(N)

Other Queue functions (cont.)

```
template<class ItemType>
QueueType<ItemType>::~QueueType()
{
 MakeEmpty();
}
```

O(N)

Other Queue functions (cont.)

```
template<class ItemType>
bool QueueType<ItemType>::IsEmpty() const
{
 return(qFront == NULL);
}
```

O(1)

```
template<class ItemType>
bool QueueType<ItemType>::IsFull() const
{
 NodeType<ItemType>* ptr;

 ptr = new NodeType<ItemType>;
 if(ptr == NULL)
   return true;
 else {
   delete ptr;
   return false;
 }
}
```

O(1)

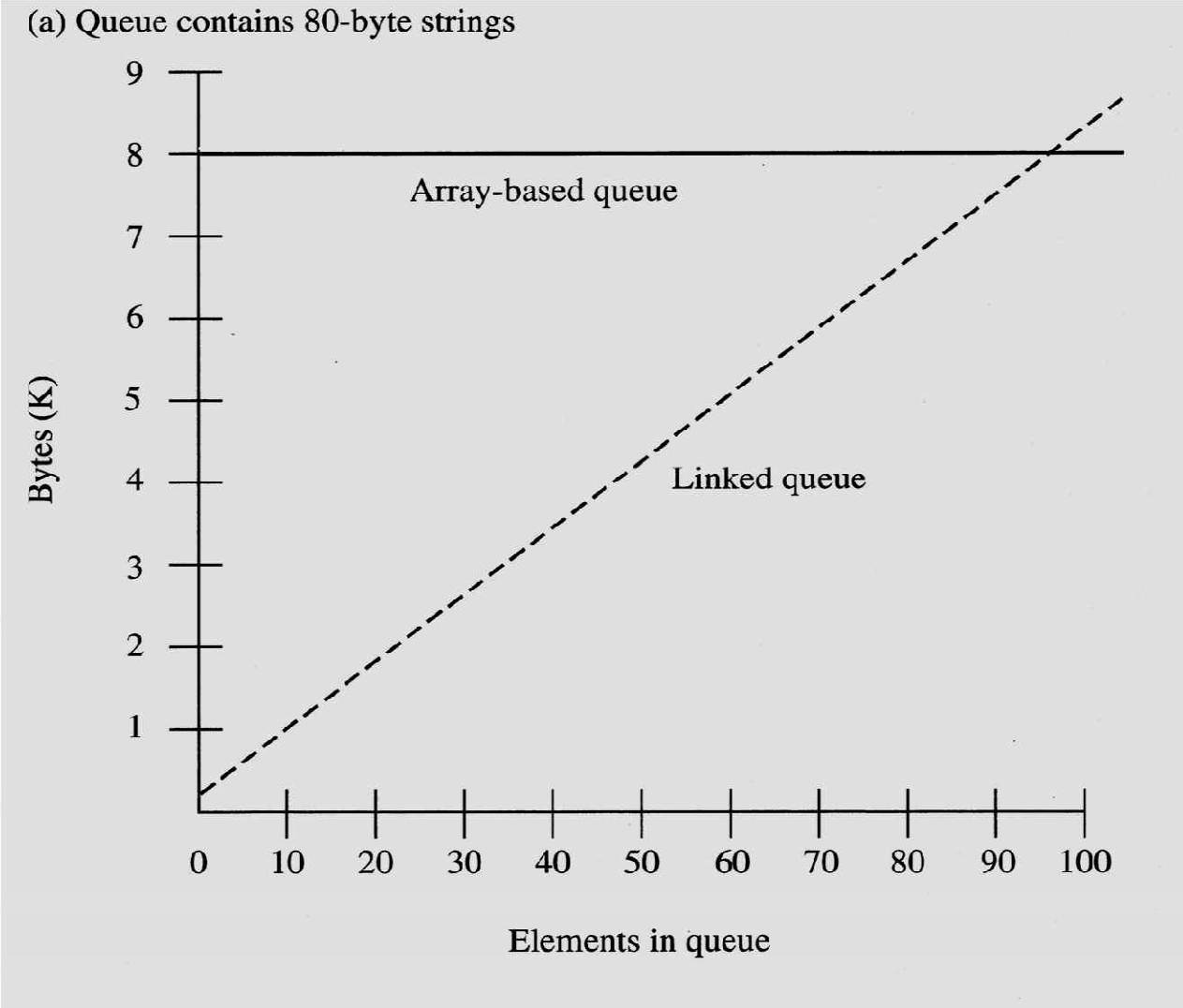
Comparing queue implementations

Big-O Comparison of Queue Operations		
Operation	Array Implementation	Linked Implementation
Class constructor	O(1)	O(1)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
IsEmpty	O(1)	O(1)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Destructor	O(1)	O(N)

Compare queue implementations in terms of memory usage

- Example 1: Assume a queue of strings
  - Array-based implementation
    - Assume a queue (size: 100) of strings (80 bytes each)
    - Assume indices take 2 bytes
    - Total memory: (80 bytes x 101 slots) + (2 bytes x 2 indices) = 8084 bytes
  - Linked-list-based implementation
    - Assume pointers take 4 bytes
    - Memory per node: 80 bytes + 4 bytes = 84 bytes

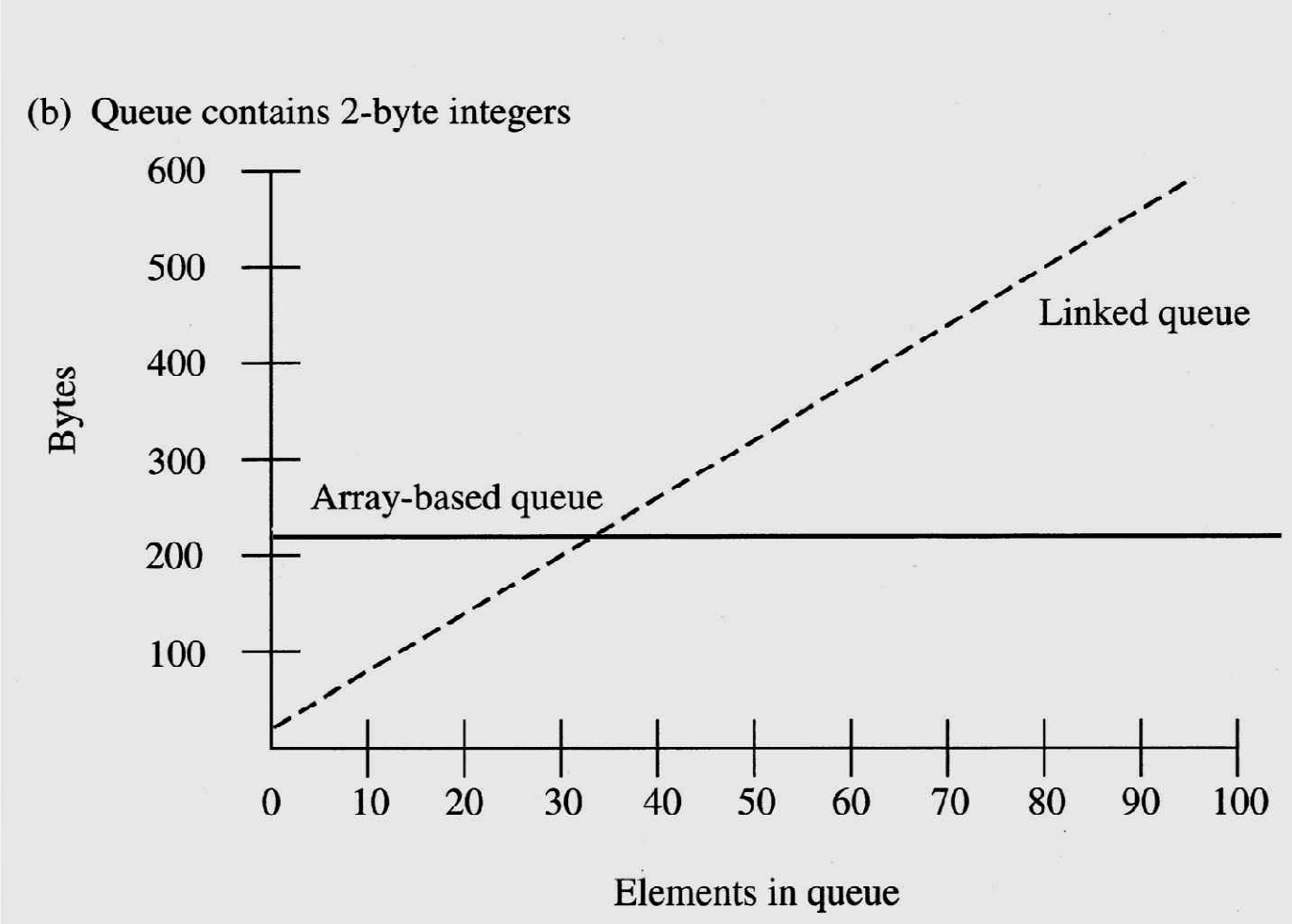
Comparing queue implementations(cont.)



Compare queue implementations in terms of memory usage(cont.)

- Example 2: Assume a queue of short integers
  - Array-based implementation
    - Assume a queue (size: 100) of short integers (2 bytes each)
    - Assume indices take 2 bytes
    - Total memory: (2 bytes x 101 slots) + (2 bytes x 2 indexes) = 206 bytes
  - Linked-list-based implementation
    - Assume pointers take 4 bytes
    - Memory per node: 2 bytes + 4 bytes = 6 bytes

Comparing queue implementations(cont.)



Array- vs Linked-list-based Queue Implementations

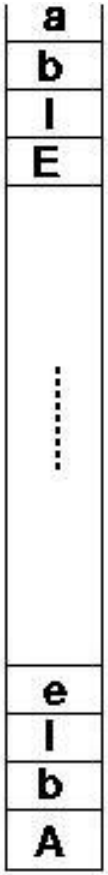
- Array-based implementation is simple but:
  - The size of the queue must be determined when the queue object is declared.
  - Space is wasted if we use less elements.
  - Cannot "enqueue" more elements than the array can hold.
- Linked-list-based queue alleviates these problems but time requirements might increase.

Example: recognizing palindromes

- A *palindrome* is a string that reads the same forward and backward.

Able was I ere I saw Elba

Example: recognizing palindromes



Stack

Queue

(1) Read the line of text into both a stack and a queue.

(2) Compare the contents of the stack and the queue character-by-character to see if they would produce the same string of characters.

Example: recognizing palindromes

```
#include <iostream.h>
#include <ctype.h>
#include "stack.h"
#include "queue.h"
int main()
{
 StackType<char> s;
 QueType<char> q;
 char ch;
 char sItem, qItem;
 int mismatches = 0;

 cout << "Enter string: "
  << endl;

 while(cin.peek() != '\n') {

   cin >> ch;
   if(isalpha(ch)) {

     if(!s.IsFull())
       s.Push(toupper(ch));

     if(!q.IsFull())
       q.Enqueue(toupper(ch));
   }
}
```

Example: recognizing palindromes

```
while( (!q.IsEmpty()) && (!s.IsEmpty()) ) {

  s.Pop(sItem);
  q.Dequeue(qItem);

  if(sItem != qItem)
    ++mismatches;
}
if (mismatches == 0)
  cout << "That is a palindrome" << endl;
else
 cout << That is not a palindrome" << endl;

return 0;
}
```

Exercise 37: Implement a client function that returns the number of items in a queue. The queue is unchanged.

int Length(QueueType& queue)

Function: Determines the number of items in the queue.
Precondition: queue has been initialized.
Postconditions: queue is unchanged

You may not assume any knowledge of how the queue is implemented.

```
int Length(QueType& queue)
{
  QueType tempQ;
  ItemType item;
  int length = 0;
  while (!queue.IsEmpty())
  {
    queue.Dequeue(item);
    tempQ.Enqueue(item);
    length++;
  }
  while (!tempQ.IsEmpty())
  {
    tempQ.Dequeue(item);
    queue.Enqueue(item);
}
  return length;
}
```

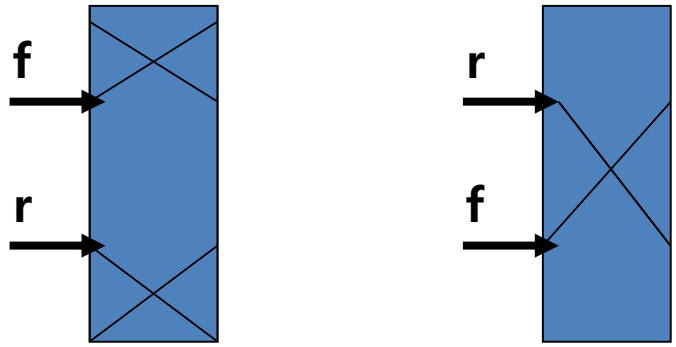
What are the time requirements using big-O?

O(N)

```
int Length(QueType& queue)
{
  QueType tempQ;
  ItemType item;
  int length = 0;
  while (!queue.IsEmpty())
  {
    queue.Dequeue(item);
    tempQ.Enqueue(item);
    length++;
  }
  while (!tempQ.IsEmpty())
  {
    tempQ.Dequeue(item);
    queue.Enqueue(item);
}
  return length;
}
```

How would you implement it as member function?

Case 1: array-based



rear - front

maxQue – (front – rear)

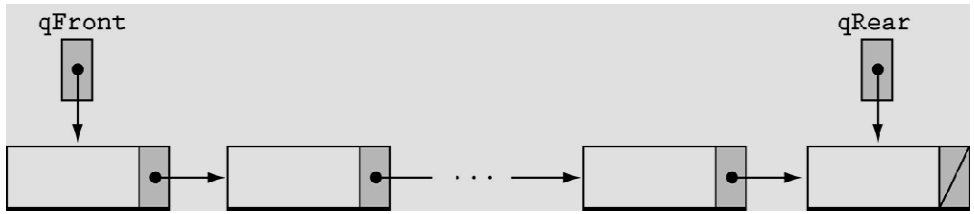
What would be the time requirements in this case using big-O?

O(1)

```
int Length(QueType& queue)
{
  QueType tempQ;
  ItemType item;
  int length = 0;
  while (!queue.IsEmpty())
  {
    queue.Dequeue(item);
    tempQ.Enqueue(item);
    length++;
  }
  while (!tempQ.IsEmpty())
  {
    tempQ.Dequeue(item);
    queue.Enqueue(item);
}
  return length;
}
```

How would you implement it as member function?

Case 2: linked-list-based



What would be the time requirements in this case using big-O?

O(N)